BLG 335E - Analysis of Algorithms I
2020/2021 Fall
Homework 3

Name: Kerem Berk Güçlü

ID: 150160058

SSH Example Command of Compile:
(main.cpp only works c++11 because of some functions) g++ -std=c++11 –o main main.cpp ./main example.csv Program works stable at Visual Studio Code with (“g++ main.cpp”)

Report

Complexity

Red–black tree

Type	tree
Invented	1972
Invented by	Rudolf Bayer

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$[1]	$O(\log n)$[1]
Insert	$O(\log n)$[1]	$O(\log n)$[1]
Delete	$O(\log n)$[1]	$O(\log n)$[1]

Inserting a key into a non-empty tree has three steps. In the first step, the BST insert operation is performed. The BST insert operation is O(height of tree) which is O(log N) because a red-black tree is balanced. The second step is to color the new node red. This step is O(1) since it just requires setting the value of one node's color field. In the third step, we restore any violated red-black properties.

Restructuring is O(1) since it involves changing at most five pointers to tree nodes. Once a restructuring is done, the insert algorithm is done, so at most 1 restructuring is done in step 3. So, in the worst-case, the restructuring that is done during insert is O(1).

Changing the colors of nodes during recoloring is O(1). However, we might then need to handle a double-red situation further up the path from the added node to the root. In the worst-case, we end up fixing a double-red situation along the entire path from the added node to the

root. So, in the worst-case, the recoloring that is done during insert is O(log N) (= time for one recoloring * max number of recoloring done = O(1) * O(log N)).

Thus, the third step (restoration of red-black properties) is O(log N) and the total time for insert is O(log N).

Search in a red-black tree is the same as any balanced binary search tree, Olog_2(n) time. Traversal is a $O(n)$ amortized operation because to search through the entire tree, you simply have to enter and exit each node.

RBT vs BST

Red-Black trees are very similar to standard BST, but with a few additional lines of code that identify a red and black node, there are a few more operations. Colored nodes allow self-balance of the data structure.

Augmenting Data Structures

First, we include the num_position[x] for i^{th} Position and position name fields in our tree nodes, respectively holding the number of position in the sub-tree rooted at **x**, including **x** itself. Implementation makes use of these fields in parallel with the **size[x]** field in the **OS_SELECT**.

size[*x*] = *size*[*left*[*x*]] + *size*[*right*[*x*]] + 1

OS-SELECT(*x*, *i*) $\triangleright i^{th}$ smallest element in the subtree rooted at *x*

k←*size*[*left*[*x*]] + 1$\triangleright$*k* = rank(*x*)

if *i*= *k* **then return** *x*

if *i*< *k*

then return OS-SELECT(*left*[*x*], *i*)

else return OS-SELECT(*right*[*x*], *i* –*k*)

Strategy: Updating subtree sizes when inserting. Also rb-insert() may need to modify the red-black tree in order maintain balance. Rotations fix up subtree sizes in O(1) time.